The Eigen C++ template library for linear algebra

April 15, 2016

Eigen

Introduction to

Eigen 3

Useful web links

- `http://eigen.tuxfamily.org`

Eigen

Eigen

The Eigen library is a linear algebra library that works with dense and sparse matrices. With the use of advanced programming techniques it performs algebraic operations with very high efficiency. It provides algorithms for the solution of linear systems, factorizations, and also interfaces to external libraries that perform linear algebra operations.

It is widely used as a library in many applications, i.e. at Google.

Eigen

Introduction

```
#include <iostream>
#include <Eigen/Dense>
int main()
{
 Eigen::MatrixXd m(2,2);
 m(0,0) = 3;
 m(1,0) = 2.5;
 m(0,1) = -1;
 m(1,1) = m(1,0) + m(0,1);
 std::cout << m << std::endl;
}
```

Eigen

(Dense) Matrices

Generic matrix

```
Matrix<typename Scalar, int RowsAtCompileTime,
                        int ColsAtCompileTime>
```

Special cases

- Matrix with fixed dimensions

  ```
  typedef Matrix<float, 4, 4> Matrix4f;
  ```

- Matrix with non-fixed dimensions

  ```
  typedef Matrix<double, Dynamic, Dynamic> MatrixXd;
  ```

- Matrix with only one fixed dimension

  ```
  typedef Matrix<int, Dynamic, 1> VectorXi;
  ```

Eigen

Initialization

- non-initialized matrix

```
MatrixXd m(2,2);
```

- initialized matrix

```
Matrix3f m;
m << 1, 2, 3, 4, 5, 6, 7, 8, 9;
```

- initialized matrix

```
MatrixXf m = MatrixXf::Constant(3, 3, 1.2);
```

Eigen

Arithmetics

Eigen implements all the arithmetic operators, and also

- transpose `a.transpose()`
- conjugate `a.conjugate()`
- adjoint `a.adjoint()`
- dot product `a.dot(b)`
- cross product `a.cross(b)`
- ...

Eigen

Expression Templates

the code

```
VectorXf a(50), b(50), c(50), d(50);
...
a = 3*b + 4*c + 5*d;
```

is implemented in such a way that it is equivalent to the execution of

```
for(int i = 0; i < 50; ++i)
  a[i] = 3*b[i] + 4*c[i] + 5*d[i];
```

just like the most performant (and optimizable by the compiler) c code.

Eigen

Advantages & disadvantages

the efficiency of the code greatly improves, but expression of the type

```
a = a.transpose();
```

are not allowed. Furthermore, compile time errors and execution time errors become mostly unreadable.

Eigen

code

```
#include <iostream>
#include <Eigen/Dense>
int main() {
  Eigen::Matrix3f m = Eigen::Matrix3f::Random();
  std::cout << "m =" << std::endl << m ; }
```

compilation

Since Eigen is a pure header file library, we need to add its path in the compilation phase with the parameter `-I`

```
g++ -I/path/to/eigen/ my_program.cpp -o my_program -O2
```

in the virtual machine eigen is available as a module

Column-major and row-major storage I

We say that a matrix is stored in row-major order if it is stored row by row. The entire first row is stored first, followed by the entire second row, and so on. Consider for example the matrix. it E.g.:

$$A = \begin{bmatrix} 8 & 2 & 2 & 9 \\ 9 & 1 & 4 & 4 \\ 3 & 5 & 4 & 5 \end{bmatrix}$$

Column major storage

[8 9 3 2 1 5 2 4 4 9 4 5]

Row major storage

[8 2 2 9 9 1 4 4 3 5 4 5]

Column-major and row-major storage II

Example:

```
Matrix<int, 3, 4, ColMajor> Acolmajor;
Acolmajor << 8, 2, 2, 9,
             9, 1, 4, 4,
             3, 5, 4, 5;
cout << "The matrix A:" << endl;
cout << Acolmajor << endl << endl;
cout << "In memory (column-major):" << endl;
for (int i = 0; i < Acolmajor.size(); i++)
  cout << *(Acolmajor.data() + i) << "  ";
cout << endl << endl;
Matrix<int, 3, 4, RowMajor> Arowmajor = Acolmajor;
cout << "In memory (row-major):" << endl;
for (int i = 0; i < Arowmajor.size(); i++)
  cout << *(Arowmajor.data() + i) << "  ";
cout << endl;
```

Column-major and row-major storage III

Output:

```
The matrix A:
8 2 2 9
9 1 4 4
3 5 4 5

In memory (column-major):
8 9 3 2 1 5 2 4 4 9 4 5

In memory (row-major):
8 2 2 9 9 1 4 4 3 5 4 5
```

Interfacing with raw buffers: the Map class I

Eigen allows to work with ”raw” C/C++ arrays using the Map class.
A Map object has a type defined by its Eigen equivalent:

Map<Matrix<**typename** Scalar, **int** RowsAtCompileTime, **int** ColsAtCompileTime> >

Example:

```
typedef Matrix<float,1,Dynamic> MatrixType;
typedef Map<MatrixType> MapType;
typedef Map<const MatrixType> MapTypeConst;   // a read-only map
const int n_dims = 5;

MatrixType m1(n_dims), m2(n_dims);
m1.setRandom();
m2.setRandom();
float *p = &m2(0);  // get the address storing the data for m2
MapType m2map(p,m2.size());   // m2map shares data with m2
MapTypeConst m2mapconst(p,m2.size());  // a read-only accessor for m2
cout << "m1: " << m1 << endl;
cout << "m2: " << m2 << endl;
cout << "Squared euclidean distance: " << (m1-m2).squaredNorm() << endl;
cout << "Squared euclidean distance, using map: " <<
  (m1-m2map).squaredNorm() << endl;
m2map(3) = 7;   // this will change m2, since they share the same array
cout << "Updated m2: " << m2 << endl;
cout << "m2 coefficient 2, constant accessor: " << m2mapconst(2) << endl;
/* m2mapconst(2) = 5; */   // this yields a compile-time error
```

Exercises

- Re-implement the fem1d code using Eigen/Dense
  - Version 1: use an Eigen::Map object to implement matrix::solve and matrix **operator**$*$ (matrix&, matrix&)
  - version 2: get rid of the matrix class and use Eigen::Matrix instead.